Analytic Sensitivity and Uncertainty Computations in Large-Scale Applications via Automatic Differentiation

Eric Phipps

Optimization & Uncertainty Quantification Department
Sandia National Laboratories
Albuquerque, NM USA

Analytic Derivatives Enable Robust Simulation and Design Capabilities

- We need analytic first & higher derivatives for predictive simulations
  - Computational design, optimization and parameter estimation
  - Stability analysis
  - Uncertainty quantification
  - Verification and validation
- Analytic derivatives improve robustness and efficiency
  - Very hard to make finite differences accurate
- Infeasible to expect application developers to code analytic derivatives
  - Time consuming, error prone, and difficult to verify
  - Thousands of possible parameters in a large code
  - Developers must understand what derivatives are needed
- Automatic differentiation solves these problems

Sandia National Laboratories

What is Automatic Differentiation (AD)?

- Technique to compute analytic derivatives without hand-coding the derivative computation
- How does it work -- freshman calculus
  - Computations are composition of simple operations (+, *, sin(), etc…) with known derivatives
  - Derivatives computed line-by-line, combined via chain rule
- Derivatives accurate as original computation
  - No finite-difference truncation errors
- Provides analytic derivatives without the time and effort of hand-coding them

$$y = \sin(e^x + x \log x), \quad x = 2$$

	x	$\frac{d}{dx}$
$x \leftarrow 2$ $\quad \frac{dx}{dx} \leftarrow 1$	2.000	1.000
$t \leftarrow e^x$ $\quad \frac{dt}{dx} \leftarrow t\frac{dx}{dx}$	7.389	7.389
$u \leftarrow \log x$ $\quad \frac{du}{dx} \leftarrow \frac{1}{x}\frac{dx}{dx}$	0.301	0.500
$v \leftarrow xu$ $\quad \frac{dv}{dx} \leftarrow u\frac{dx}{dx} + x\frac{du}{dx}$	0.602	1.301
$w \leftarrow t + v$ $\quad \frac{dw}{dx} \leftarrow \frac{dt}{dx} + \frac{dv}{dx}$	7.991	8.690
$y \leftarrow \sin w$ $\quad \frac{dy}{dx} \leftarrow \cos(w)\frac{dw}{dx}$	0.991	-1.188

Sandia National Laboratories

AD Takes Two Basic Forms

$$x \in \mathrm{R}^n,\ f : \mathrm{R}^n \to \mathrm{R}^m,\ y = f(x) \in \mathrm{R}^m$$

- Forward Mode:
  - Propagate derivatives of intermediate variables w.r.t. independent variables *forward*

$$c = \varphi(a,b) \implies \frac{\partial c}{\partial x_i} = \frac{\partial \varphi}{\partial a}\frac{\partial a}{\partial x_i} + \frac{\partial \varphi}{\partial b}\frac{\partial b}{\partial x_i}$$

  - Change of variables

$$x = Vz,\quad V \in \mathrm{R}^{n\times p} \implies \frac{\partial y}{\partial z} = \frac{\partial f}{\partial x}V$$

  - Complexity

$$\text{ops}\left(f,\ \frac{\partial f}{\partial x}V\right) \approx (1 + 1.5p)\text{ops}(f)$$

- Reverse Mode:
  - Propagate derivatives of dependent variables w.r.t. intermediate variables *backwards*

$$c = \varphi(a,b) \implies \frac{\partial y_j}{\partial a} = \frac{\partial y_j}{\partial c}\frac{\partial \varphi}{\partial a},\quad \frac{\partial y_j}{\partial b} = \frac{\partial y_j}{\partial c}\frac{\partial \varphi}{\partial b}$$

  - Change of variables

$$z = W^T y,\quad W \in \mathrm{R}^{m\times q} \implies \frac{\partial z}{\partial x} = W^T\frac{\partial f}{\partial x}$$

  - Complexity

$$\text{ops}\left(f,\ W^T\frac{\partial f}{\partial x}\right) \approx (1.5 + 2.5q)\text{ops}(f)$$

Sandia National Laboratories

How is AD Implemented?

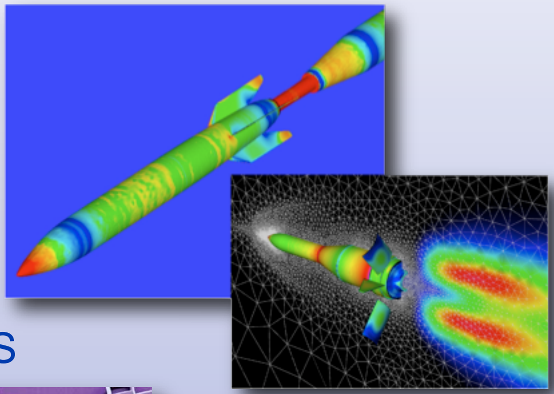
- Source transformation
  - Preprocessor implements AD
  - Very efficient derivative code
  - Works well for FORTRAN, some C
  - Extremely difficult for C++
  - OpenAD, ADIFOR, ADIC (Argonne National Lab & Rice University)
- Operator overloading
  - All intrinsic operations/elementary operations overloaded for AD data types
  - Change data types in code from floats/doubles to AD types
    - C++ templates greatly help
  - Easy to incorporate into C++ codes
  - Slower than source transformation due to function call overhead
  - ADOL-C (TU-Desden), TFAD<> (expression templates)
- Effective implementation requires appropriate tool and approach

Sandia National Laboratories

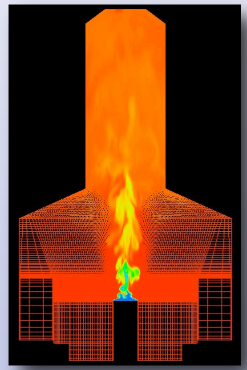
Sandia Physics Simulation Codes

- Element-based
  - Finite element, finite volume, finite difference, network, etc…
- Large-scale
  - Billions of unknowns
- Parallel
  - MPI-based SPMD
  - Distributed memory
- C++
  - Object oriented
  - Some coupling to legacy Fortran libraries
- We need AD techniques that fit these requirements

Fluids

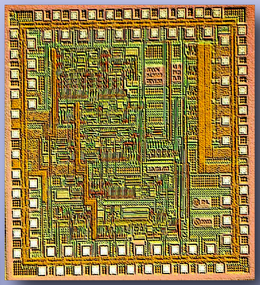


Combustion

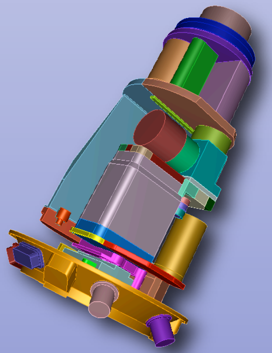


MEMS

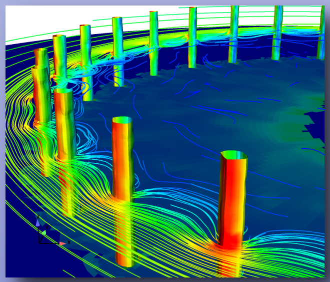
Circuits



Structures



Plasmas



Sacado: AD Tools for C++ Applications

- Package in Trilinos 8.0
- Implements several modes of AD
  - Forward (Jacobians, Jacobian-vector products, …)
  - Reverse (Gradients, Jacobian-transpose-vector products, …)
  - Taylor (High-order univariate Taylor series)
- AD via operator overloading and C++ templating
  - Expression templates for OO efficiency
  - Application code templating for easy incorporation
- Designed for use in large-scale C++ codes
  - Apply AD at “element-level”
  - Manually integrate derivatives into parallel data structures and solvers
  - `Sacado::FEApp` example demonstrates approach

Simple Sacado Example

```
// The function to differentiate

double func(double a, double b, double c) {
  double r = c*std::log(b+1.)/std::sin(a);
  return r;
}

int main(int argc, char **argv) {
  double a = std::atan(1.0);                       // pi/4
  double b = 2.0;
  double c = 3.0;





  // Compute function
  double r = func(a, b, c);
```

Simple Sacado Example

```
#include "Sacado.hpp"

// The function to differentiate
template <typename ScalarT>
ScalarT func(const ScalarT& a, const ScalarT& b, const ScalarT& c) {
  ScalarT r = c*std::log(b+1.)/std::sin(a);
  return r;
}

int main(int argc, char **argv) {
  double a = std::atan(1.0);                       // pi/4
  double b = 2.0;
  double c = 3.0;

  int num_deriv = 2;                               // Number of independent variables
  Sacado::Fad::DFad<double> afad(num_deriv, 0, a); // First (0) indep. var
  Sacado::Fad::DFad<double> bfad(num_deriv, 1, b); // Second (1) indep. var
  Sacado::Fad::DFad<double> cfad(c);               // Passive variable

  // Compute function
  double r = func(a, b, c);

  // Compute function and derivative with AD
  Sacado::Fad::DFad<double> rfad = func(afad, bfad, cfad);

  // Extract value and derivatives
  double r_ad = rfad.val();     // r
  double drda_ad = rfad.dx(0);  // dr/da
  double drdb_ad = rfad.dx(1);  // dr/db
```

Differentiating Element-Based Production Applications

- Global residual computation (ignoring boundary computations):

$$f(\dot{x}, x, t, p) = \sum_{i=1}^{N} Q_i^T e_{k_i}(P_i\dot{x}, P_i x, t, p)$$

- Time-step Jacobian computation:

$$\alpha\frac{\partial f}{\partial \dot{x}} + \beta\frac{\partial f}{\partial x} = \sum_{i=1}^{N} Q_i^T \left(\alpha\frac{\partial e_{k_i}}{\partial \dot{x}_i} + \beta\frac{\partial e_{k_i}}{\partial x_i}\right) P_i, \quad \dot{x}_i = P_i\dot{x}, \;\; x_i = P_i x$$

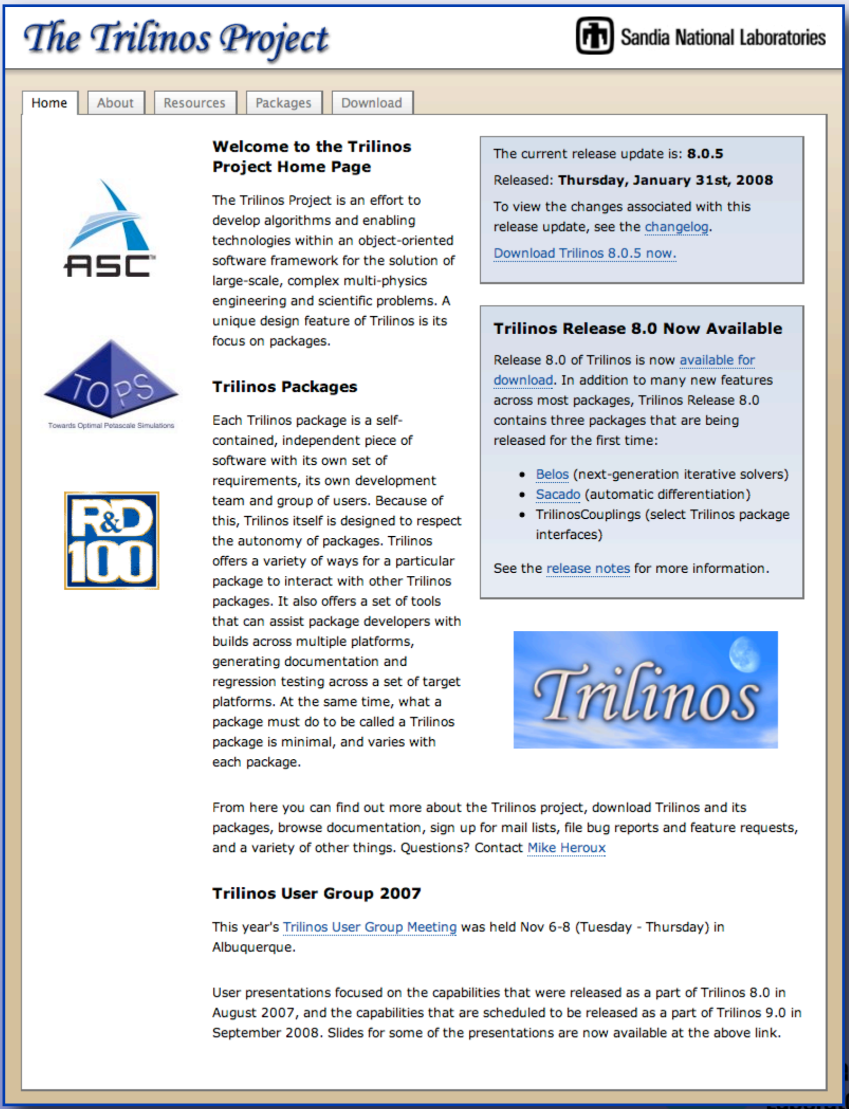
- Parameter derivative computation:

$$\frac{\partial f}{\partial p} = \sum_{i=1}^{N} Q_i^T \frac{\partial e_{k_i}}{\partial p}$$

- Hybrid symbolic/AD procedure

The Trilinos Project

- http://trilinos.sandia.gov
- Algorithms and enabling technologies
- Large-scale scientific and engineering applications
- Object oriented framework
- “String of Pearls”
- *Focus on packages*
  - Over 30 packages in 8.0 release
  - Over 40 in development



Welcome to the Trilinos Project Home Page

The Trilinos Project is an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems. A unique design feature of Trilinos is its focus on packages.

Trilinos Packages

Each Trilinos package is a self-contained, independent piece of software with its own set of requirements, its own development team and group of users. Because of this, Trilinos itself is designed to respect the autonomy of packages. Trilinos offers a variety of ways for a particular package to interact with other Trilinos packages. It also offers a set of tools that can assist package developers with builds across multiple platforms, generating documentation and regression testing across a set of target platforms. At the same time, what a package must do to be called a Trilinos package is minimal, and varies with each package.

The current release update is: **8.0.5**

Released: **Thursday, January 31st, 2008**

To view the changes associated with this release update, see the changelog.

Download Trilinos 8.0.5 now.

Trilinos Release 8.0 Now Available

Release 8.0 of Trilinos is now available for download. In addition to many new features across most packages, Trilinos Release 8.0 contains three packages that are being released for the first time:

- Belos (next-generation iterative solvers)
- Sacado (automatic differentiation)
- TrilinosCouplings (select Trilinos package interfaces)

See the release notes for more information.

From here you can find out more about the Trilinos project, download Trilinos and its packages, browse documentation, sign up for mail lists, file bug reports and feature requests, and a variety of other things. Questions? Contact Mike Heroux

Trilinos User Group 2007

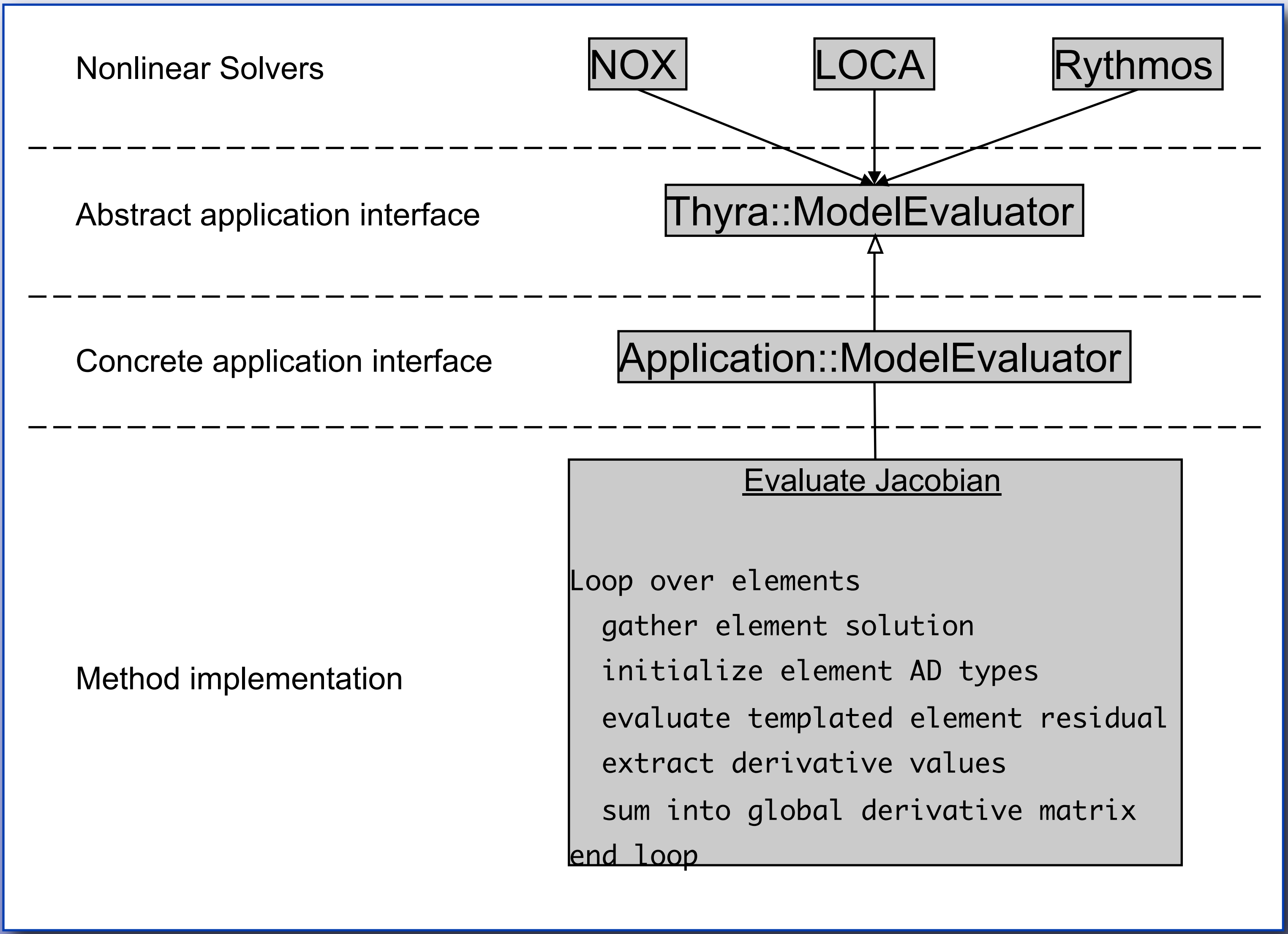
This year's Trilinos User Group Meeting was held Nov 6-8 (Tuesday - Thursday) in Albuquerque.

User presentations focused on the capabilities that were released as a part of Trilinos 8.0 in August 2007, and the capabilities that are scheduled to be released as a part of Trilinos 9.0 in September 2008. Slides for some of the presentations are now available at the above link.

Trilinos Packages

	Objective	Package(s)
Discretizations	Meshing & Spatial Discretizations	phdMesh, Intrepid
	Time Integration	Rythmos
Methods	Automatic Differentiation	Sacado
	Mortar Methods	Moertel
Core	Linear algebra objects	Epetra, Jpetra, Tpetra
	Abstract interfaces	Thyra, Stratimikos, RTOp
	Load Balancing	Zoltan, Isorropia
	“Skins”	PyTrilinos, WebTrilinos, Star-P, *ForTrilinos*
	C++ utilities, (some) I/O	Teuchos, EpetraExt, Kokkos, Triutils
Solvers	Iterative (Krylov) linear solvers	AztecOO, Belos, Komplex
	Direct sparse linear solvers	Amesos
	Direct dense linear solvers	Epetra, Teuchos, Pliris
	Iterative eigenvalue solvers	Anasazi
	ILU-type preconditioners	AztecOO, IFPACK
	Multilevel preconditioners	ML, CLAPS
	Block preconditioners	Meros
	Nonlinear system solvers	NOX, LOCA
	Optimization (SAND)	MOOCHO, Aristos

Integrating AD with Trilinos Solver Capabilities



```
Loop over elements
  gather element solution
  initialize element AD types
  evaluate templated element residual
  extract derivative values
  sum into global derivative matrix
end loop
```

Scalability of This Approach

Set of N hypothetical chemical species:

$$2X_j \rightleftharpoons X_{j-1} + X_{j+1}, \quad j = 2, \ldots, N-1$$

Steady-state mass transfer equations:

$$\mathbf{u} \cdot \nabla Y_j + \nabla^2 Y_j = \dot{\omega}_j, \quad j = 1, \ldots, N-1$$

$$\sum_{j=1}^{N} Y_j = 1$$

Forward mode AD

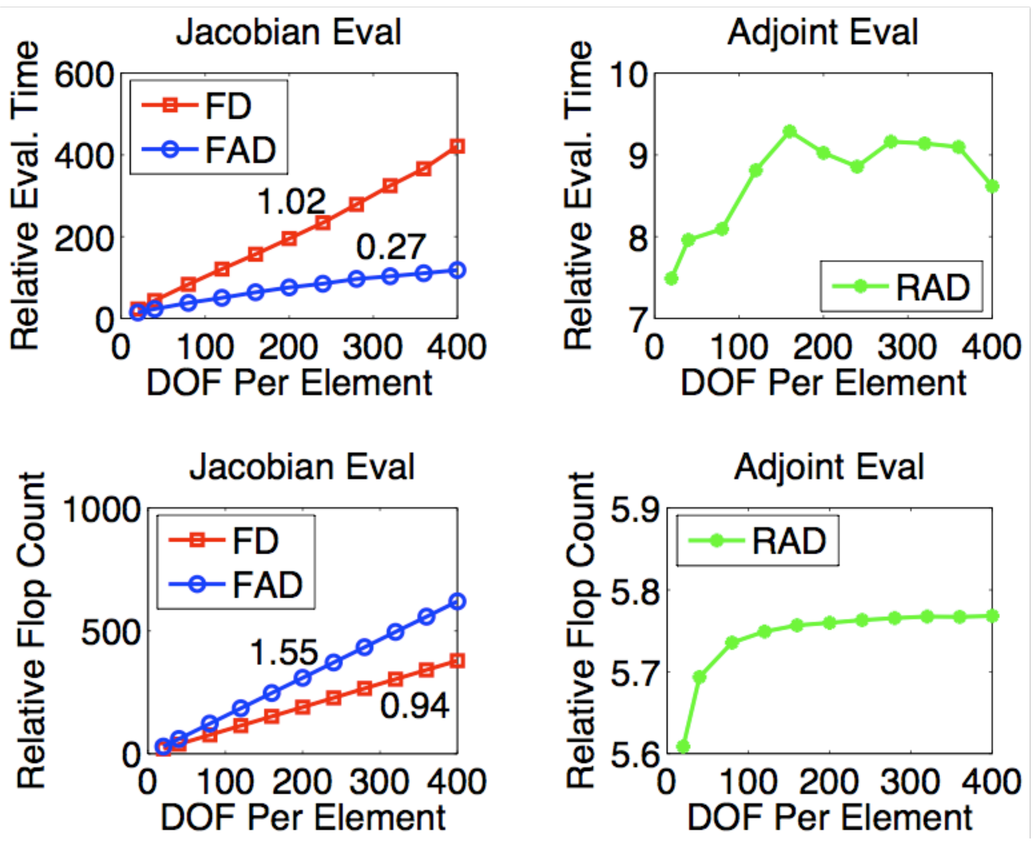
- ✓Faster than FD
- ✓Better scalability in number of PDEs
- ✓Analytic Derivative

Reverse mode AD

- ✓Scalable adjoint/gradient

$$J^T w = \nabla(w^T f(x))$$

Scalability of the element-level derivative computation

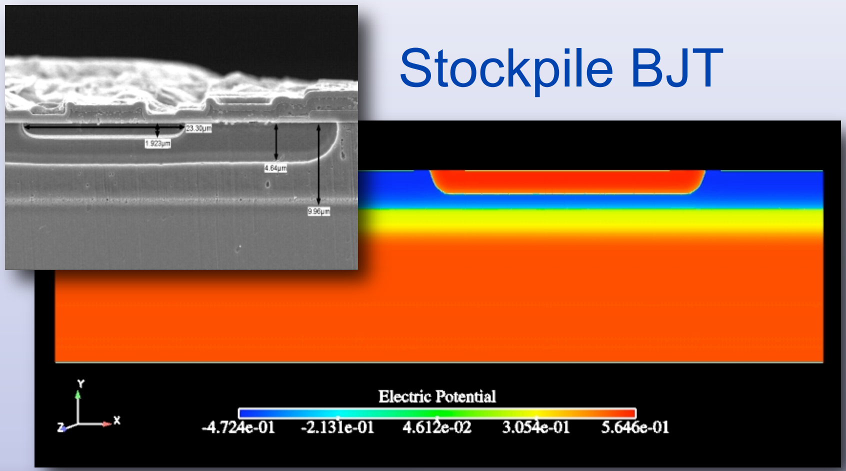


DOF per element = 4*N

QASPR

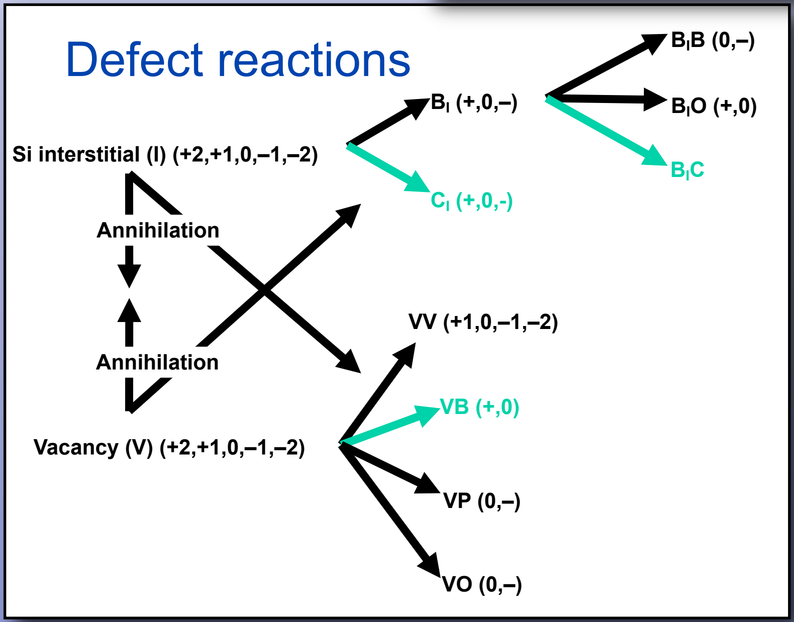
Qualification of electronic devices in hostile environments

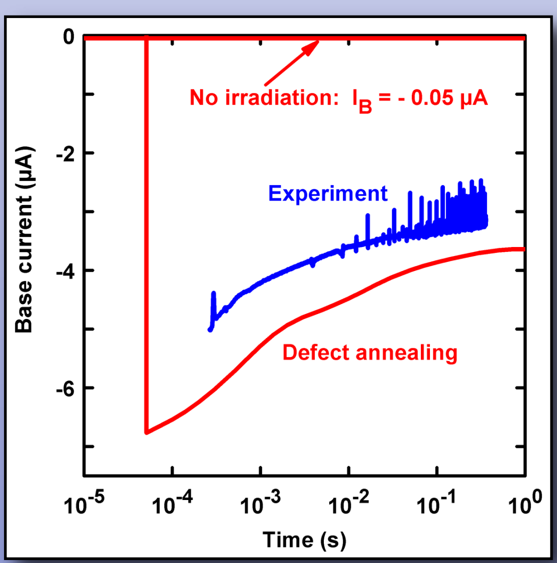
Stockpile BJT



CHARON

PDE semiconductor device simulation





Sandia National Laboratories

Charon Drift-Diffusion Formulation with Defects

Current Conservation for e- and h+

$$\frac{\partial n}{\partial t} - \nabla \cdot J_n = -R_n(\psi, n, p, Y_1, \ldots, Y_N), \quad J_n = -n\mu_n \nabla\psi + D_n \nabla n$$

$$\frac{\partial p}{\partial t} + \nabla \cdot J_p = -R_p(\psi, n, p, Y_1, \ldots, Y_N), \quad J_p = -p\mu_p \nabla\psi - D_p \nabla p$$

Defect Continuity

$$\frac{\partial Y_i}{\partial t} + \nabla \cdot J_{Y_i} = -R_{Y_i}(\psi, n, p, Y_1, \ldots, Y_N), \quad J_{Y_i} = -\mu_i Y_i \nabla\psi - D_i \nabla Y_i$$

Electric potential

$$-\nabla(\varepsilon\nabla\psi(x)) = -q\left(p(x) - n(x) + N_D^+(x) - N_A^-(x)\right) - \sum_{i=1}^{N} q_i Y_i(x)$$

Recombination/ generation source terms R_X

Include electron capture and hole capture by defect species and reactions between various defect species

Electron emission/ capture

$Z^i \leftrightarrow Z^{i+1} + e^-$

Activation Energy

$$R_{[Z^i \to Z^{i+1}+e^-]} \propto \sigma_{[Z^i \to Z^{i+1}+e^-]} Z^i \exp\left(\frac{\Delta E_{[Z^i \to Z^{i+1}+e^-]}}{kT}\right)$$

Cross section

Sandia National Laboratories

Forward Sensitivity Analysis with Rythmos

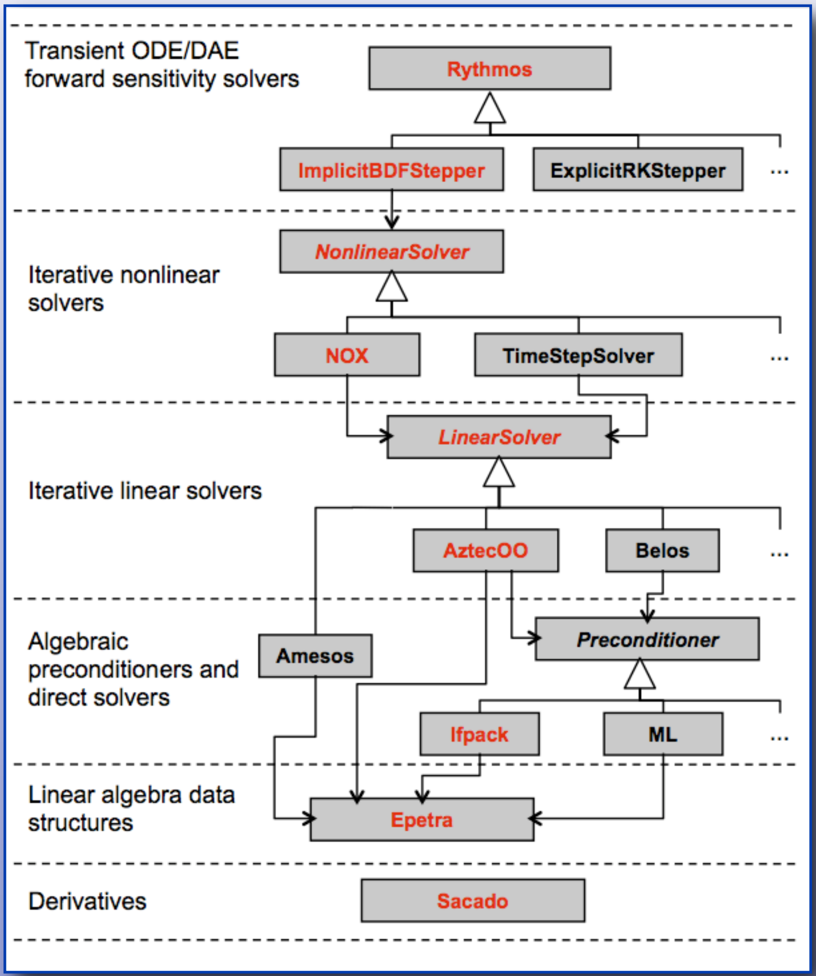
- Discretized PDE system:

$$f(\dot{x}, x, p, t) = 0$$
$$\hat{g}(p,t) = g(\dot{x}(t), x(t), p, t)$$

- Forward sensitivity problem

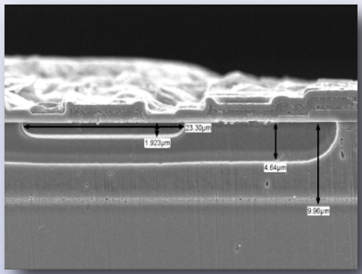
$$\frac{\partial f}{\partial \dot{x}}\left(\frac{\partial \dot{x}}{\partial p}\right) + \frac{\partial f}{\partial x}\left(\frac{\partial x}{\partial p}\right) + \frac{\partial f}{\partial p} = 0$$
$$\frac{\partial \hat{g}}{\partial p} = \frac{\partial g}{\partial \dot{x}}\frac{\partial \dot{x}}{\partial p} + \frac{\partial g}{\partial x}\frac{\partial x}{\partial p} + \frac{\partial g}{\partial p}$$

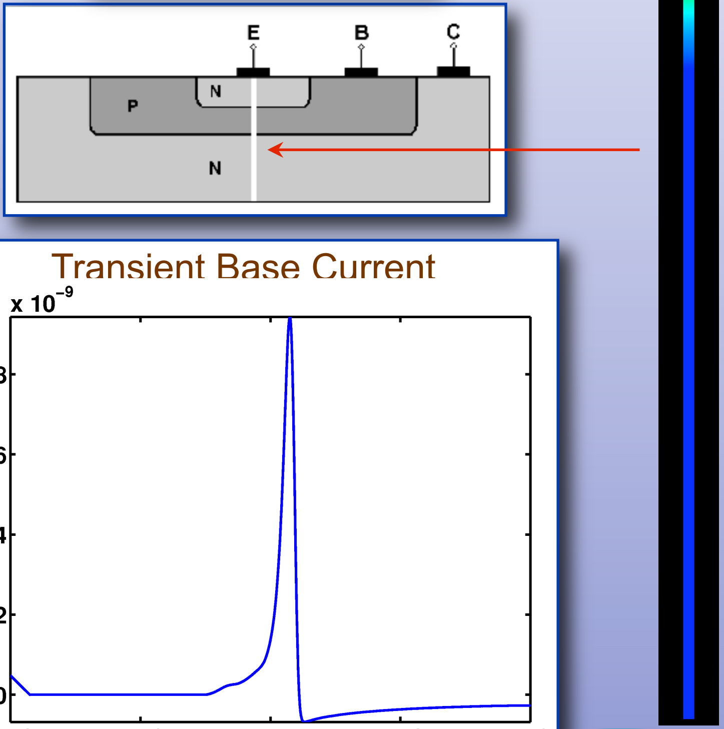
- Rythmos time integration package
  - Todd Coffey, Ross Bartlett (SNL)
  - Implicit BDF time integration method
  - Variable order, step size
  - Staggered corrector forward sensitivity method (Barton)

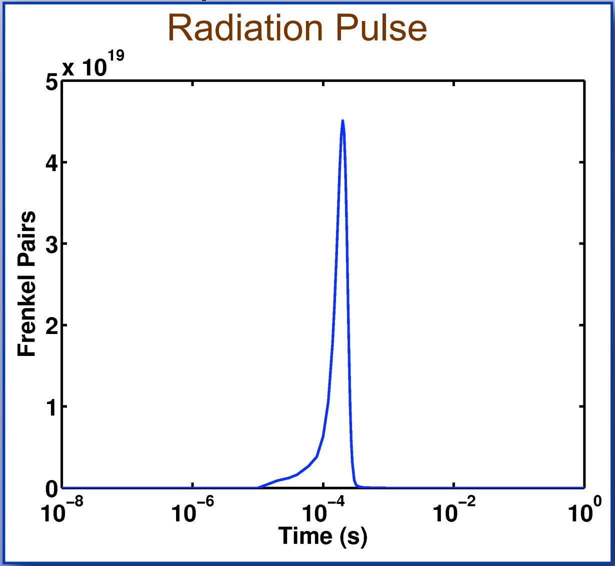


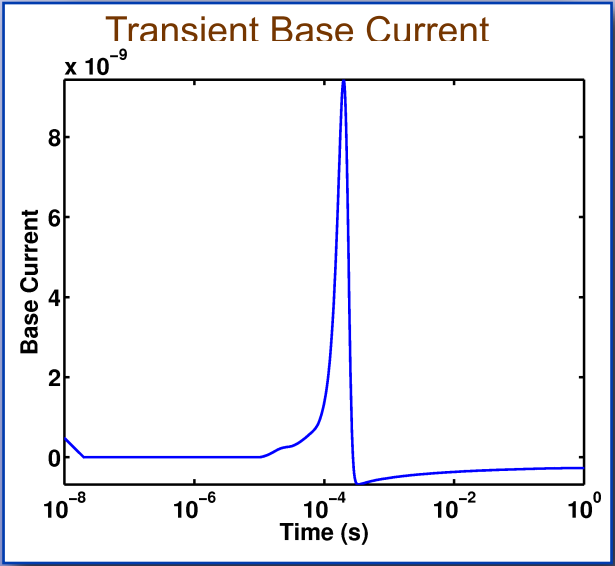
Sensitivity Analysis of a Pseudo-1D BJT

- 9x0.1 micron pseudo-1D simulation
- 1384x1 quad cells, linear finite elements + SUPG
- 2 carriers + 1 potential + 36 defect species
- 108,030 unknowns on 32 processors
- 84 carrier-defect reactions
- 126 parameters for sensitivity analysis
- AD Jacobian, parameter deriv's
- Base current provides observation function

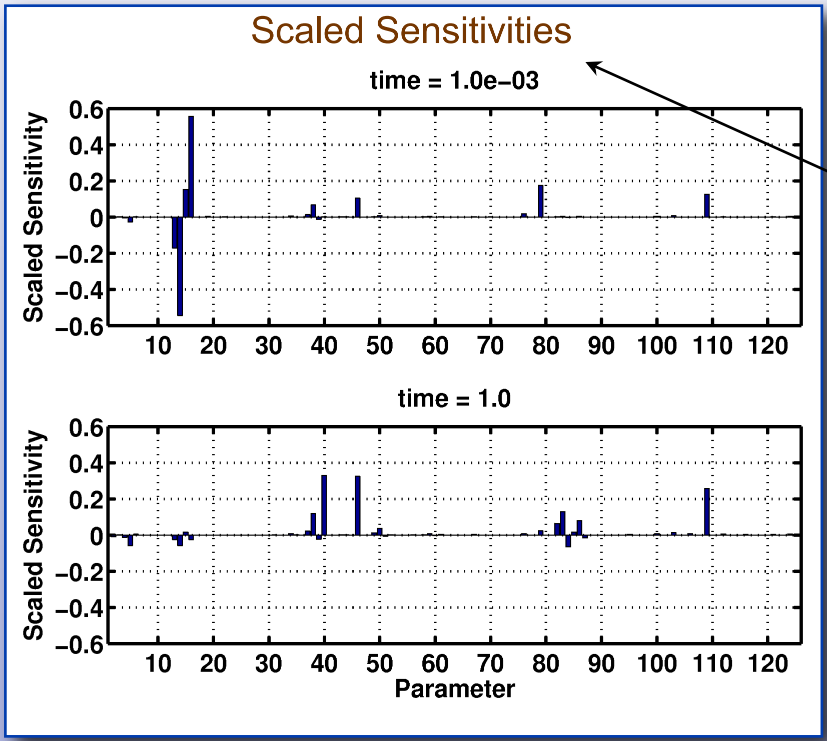




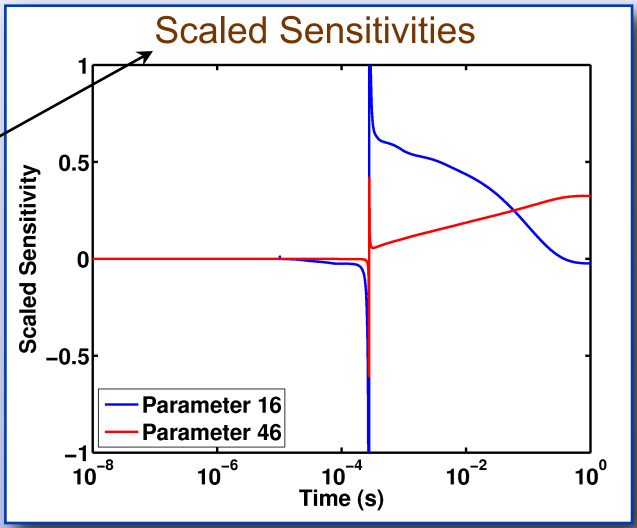


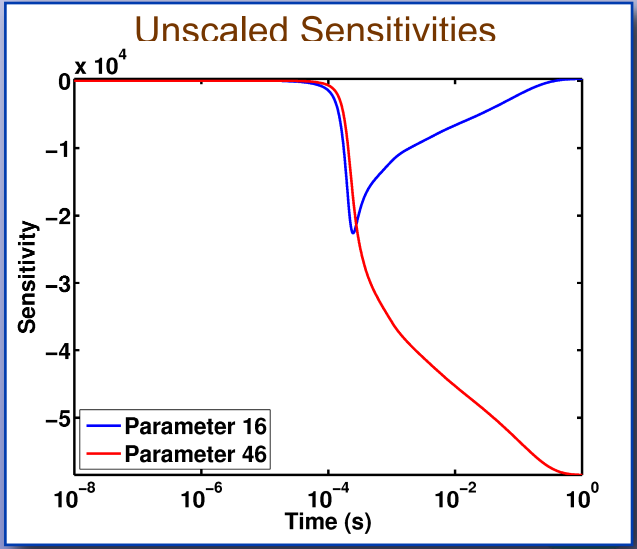


Transient Base Current Sensitivities



$$\frac{p}{I}\frac{dI}{dp}$$



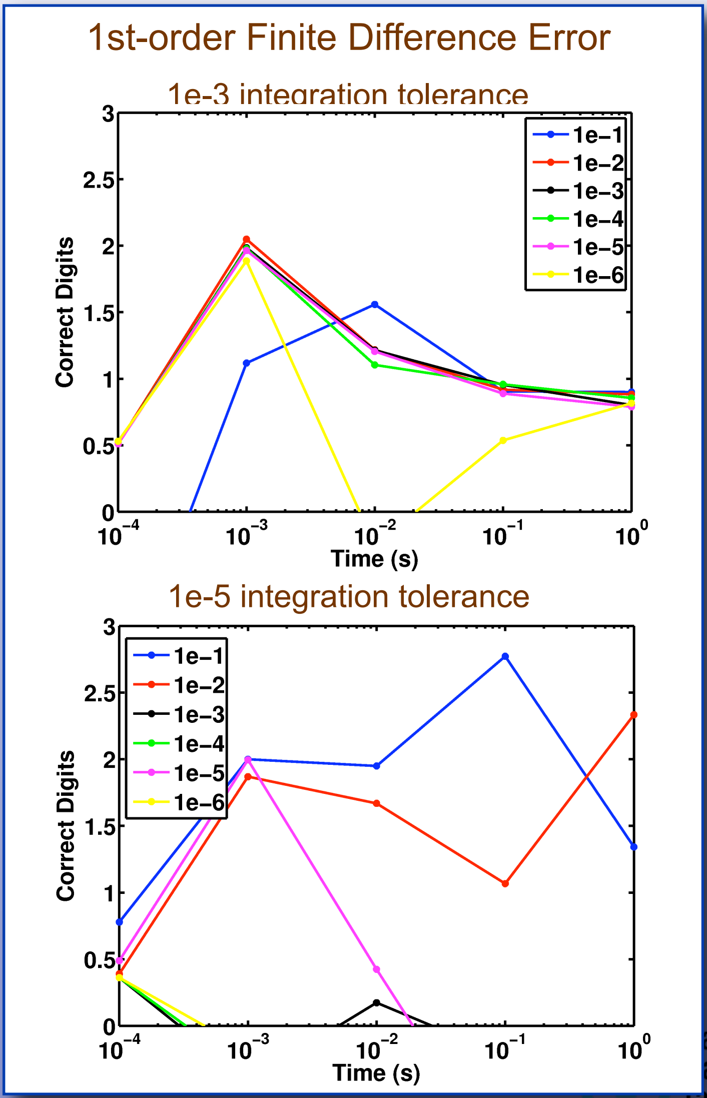


#	Reaction	Parameter	Value
14	$V^{--} \rightarrow e^- + V^-$	activation energy	0.09
16	$e^- + V^0 \rightarrow V^-$	cross-section	2.40E-14
46	$e^- + PV^0 \rightarrow PV^0$	cross-section	1.50E-15

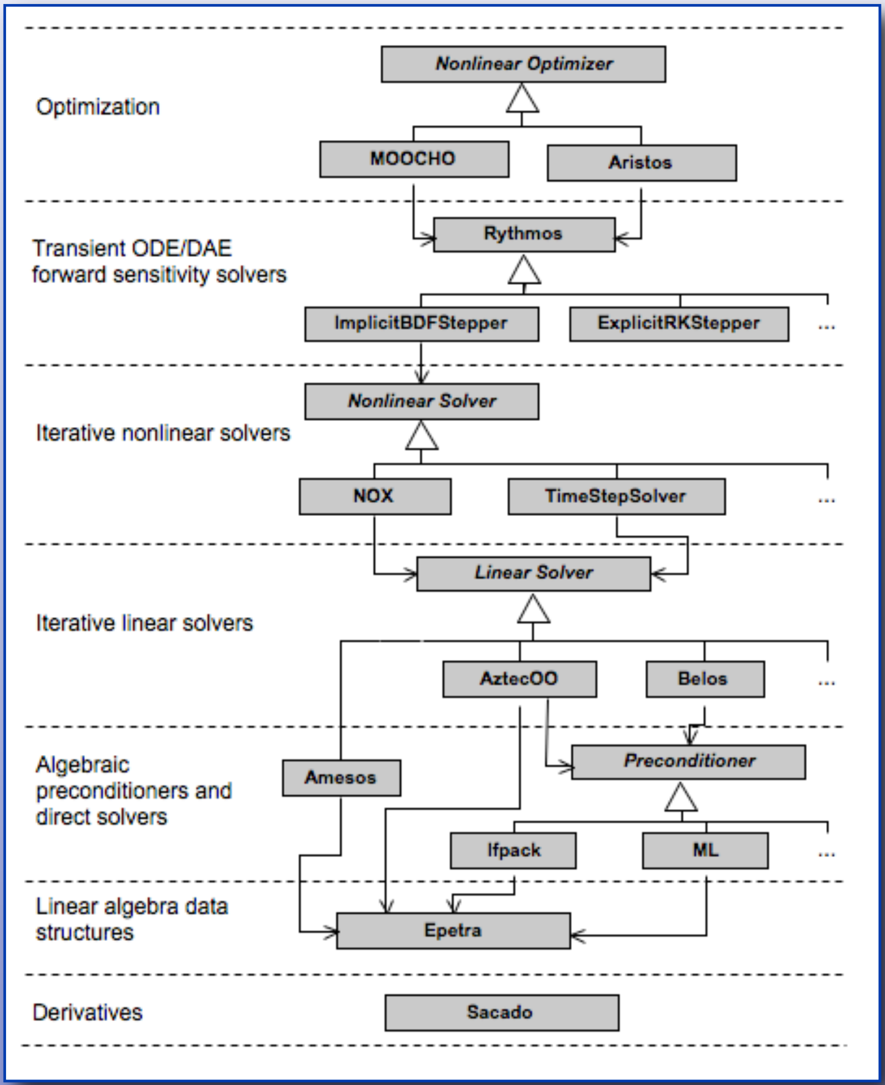
Sandia National Laboratories

Comparison to Black-Box Finite Differences

- Run-times:
  - Forward simulation: 105 min.
  - Direct sensitivities: 931 min.
  - Black-box, first-order FD: ~13,000 min.
- Direct approach more efficient
  - 14x speed-up
- Direct approach more accurate
  - 1-2 correct digits w/FD
  - FD requires tighter tolerances to achieve higher accuracy
- Direct approach more robust
  - Accuracy solely dictated by time-integration error



Stage is Set for Model Calibration



Leveraging Template Infrastructure

- Application code templating allows easy incorporation of new AD data types
  - Second derivatives
    - `Sacado::Fad::DFad< Sacado::Rad::DFad<double> >`

$$\frac{\partial}{\partial x}\left(\frac{\partial f}{\partial x}V_1\right)V_2$$

    - `Sacado::Rad::ADvar< Sacado::Rad::DFad<double> >`

$$W^T\frac{\partial}{\partial x}\left(\frac{\partial f}{\partial x}V\right)$$

  - Taylor polynomials
    - `Sacado::Tay::Taylor<double>`

$$x(t)=\sum_{k=0}^{d}x_k t^k \longrightarrow \sum_{k=0}^{d}f_k t^k = f(x(t))+O(t^{d+1}),\quad f_k=\frac{1}{k!}\frac{d^k}{dt^k}f(x(t))$$

  - Polynomial uncertainty representations

Sandia National Laboratories

Uncertainty Quantification

- Quantifying uncertainties critical for predictive simulation
- Aleatory or irreducible uncertainty: "inherent randomness"
  - Probability distribution representations
  - Monte Carlo sampling and its many variants (e.g., LHS)
  - Stochastic collocation
  - Polynomial chaos and generalized polynomial chaos
- Epistemic or reducible uncertainty: "lack of knowledge"
  - Set/interval representations
  - Interval arithmetic
  - Possibility/evidence theory
  - Probability boxes
- Aleatory uncertainty for parametric uncertainty

Sandia National Laboratories

Stochastic Galerkin Methods
(For parametric uncertainty)

- Deterministic problem (possibly after spatial discretization):

$$\text{Find } \boldsymbol{u}(\boldsymbol{p}) \text{ such that } \boldsymbol{F}(\boldsymbol{u};\boldsymbol{p}) = 0,\ \boldsymbol{p} \in \boldsymbol{\Gamma} \subset \mathbf{R}^M$$

- Stochastic problem:

$$\text{Find } \boldsymbol{u}(\boldsymbol{\xi}) \text{ such that } \boldsymbol{F}(\boldsymbol{u};\boldsymbol{\xi}) = 0,\ \boldsymbol{\xi} : \boldsymbol{\Omega} \to \boldsymbol{\Gamma}$$

- Galerkin approximation:

$$\hat{\boldsymbol{u}}(\boldsymbol{\xi}) = \sum_{i=0}^{P} \boldsymbol{u}_i \psi_i(\boldsymbol{\xi}) \to \int_{\Omega} \boldsymbol{F}(\hat{\boldsymbol{u}}(\boldsymbol{\xi});\boldsymbol{\xi})\psi_i(\boldsymbol{\xi})d\mu = 0$$

- Most methods can be obtained by choice of approximating space and basis (Gunzberger & Webster)
  - Space of piecewise constant functions: Monte Carlo
  - Space of complete polynomials of a given degree
    - Lagrange interpolation polynomial basis: Stochastic collocation
    - Hermite polynomial basis: Polynomial chaos
    - General orthogonal polynomial basis: Generalized polynomial chaos
- Choice of space/basis dictates structure of nonlinear problem

Sandia National Laboratories

SG Methods via AD

$$F_i(u_0, \ldots, u_P) \equiv \int_\Omega F(\hat{u}(\xi); \xi)\psi_i(\xi) d\mu = 0$$

- Galerkin residuals typically evaluated by quadrature
- Galerkin method can also be viewed as a projection
- Idea:
  - Given computer code to evaluate deterministic F
  - Compute projection operation by operation in evaluation of F, in an AD-like manner
  - Need way to compute projections of each operation

$$\text{Given } a = \sum_{i=0}^{P} a_i\psi_i(\xi), \quad b = \sum_{i=0}^{P} b_i\psi_i(\xi),$$

$$\text{Find } c = \sum_{i=0}^{P} c_i\psi_i(\xi) \text{ such that } \int_\Omega (c - \varphi(a,b))\psi_i d\mu = 0, \quad i = 0, \ldots P$$

  - Worked these out for orthogonal bases (e.g., polynomial chaos)

Sandia National Laboratories

Projections of Intermediate Operations

$$\langle fg \rangle \equiv \int_\Omega f(\xi) g(\xi) d\mu, \quad \{\psi_k\}_{k=0}^P \text{ orthogonal}$$

- Addition/subtraction

$$c = a \pm b \Rightarrow c_i = a_i \pm b_i$$

- Multiplication

$$c = a \times b \Rightarrow \sum_i c_i \psi_i = \sum_i \sum_j a_i b_j \psi_i \psi_j \rightarrow c_k = \sum_i \sum_j a_i b_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_k^2 \rangle}$$

- Division

$$c = a/b \Rightarrow \sum_i \sum_j c_i b_j \psi_i \psi_j = \sum_i a_i \psi_i \rightarrow \sum_i \sum_j c_i b_j \langle \psi_i \psi_j \psi_k \rangle = a_k \langle \psi_k^2 \rangle$$

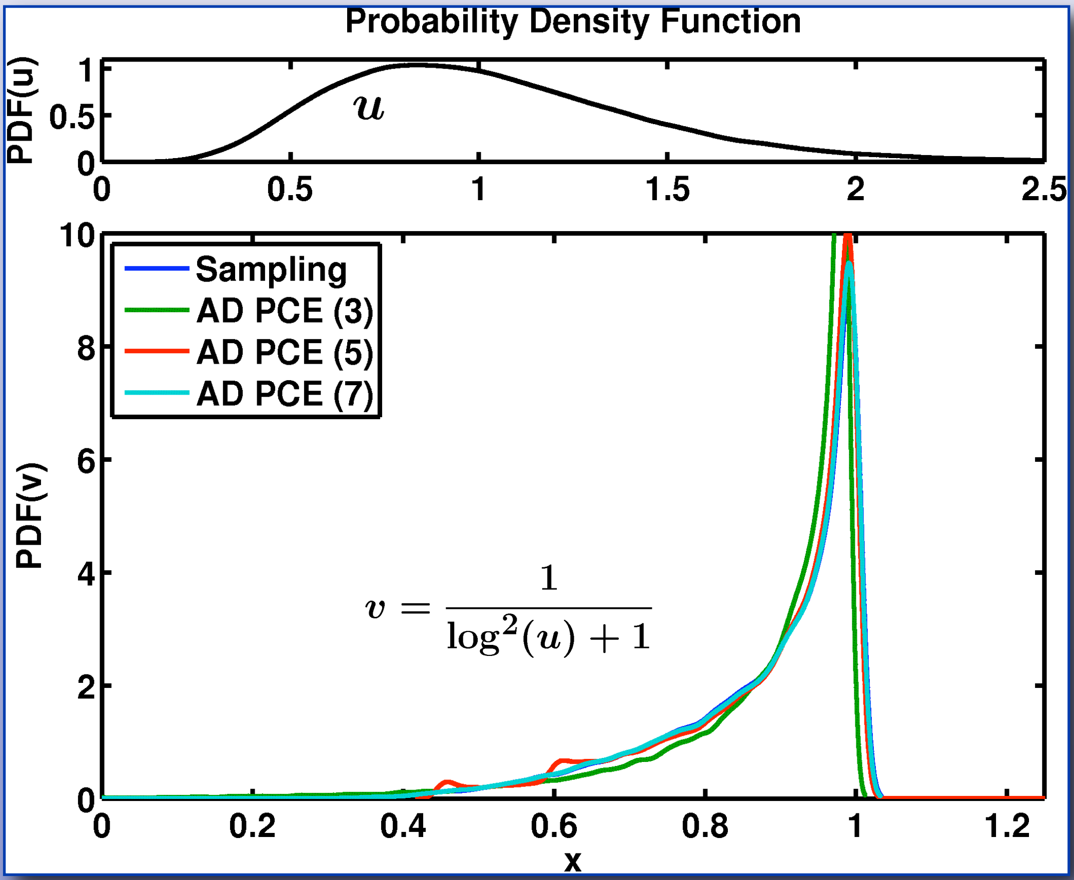
- Transcendental
  - Taylor series
  - Differential equations
- Implemented in a new Trilinos package called *Stokhos*, wrapped by Sacado for AD

Sandia National Laboratories

AD Polynomial Chaos for a Simple Function

$$u(\xi) = \psi_0(\xi) + 0.4\psi_1(\xi) + 0.06\psi_2(\xi) + 0.002\psi_3(\xi)$$

- Univariate Hermite basis
- Gaussian random variable
- Degree 3, 5, 7 PC expansion computed by AD



SG for Implicit Systems

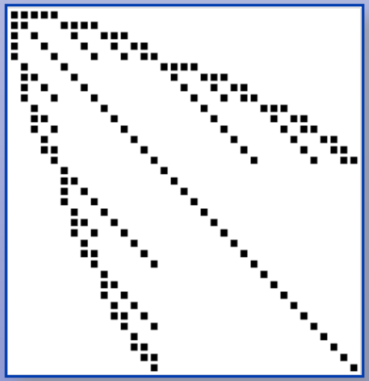
- Propagating polynomials forward through code provides stochastic residuals

$$F_i(u_0, \ldots, u_P) \equiv \int_\Omega F(\hat{u}(\xi); \xi)\psi_i(\xi) d\mu = 0$$

- For implicit systems, also need Jacobians

$$\frac{\partial F_i}{\partial u_j}, \; i, j = 0, \ldots, P$$

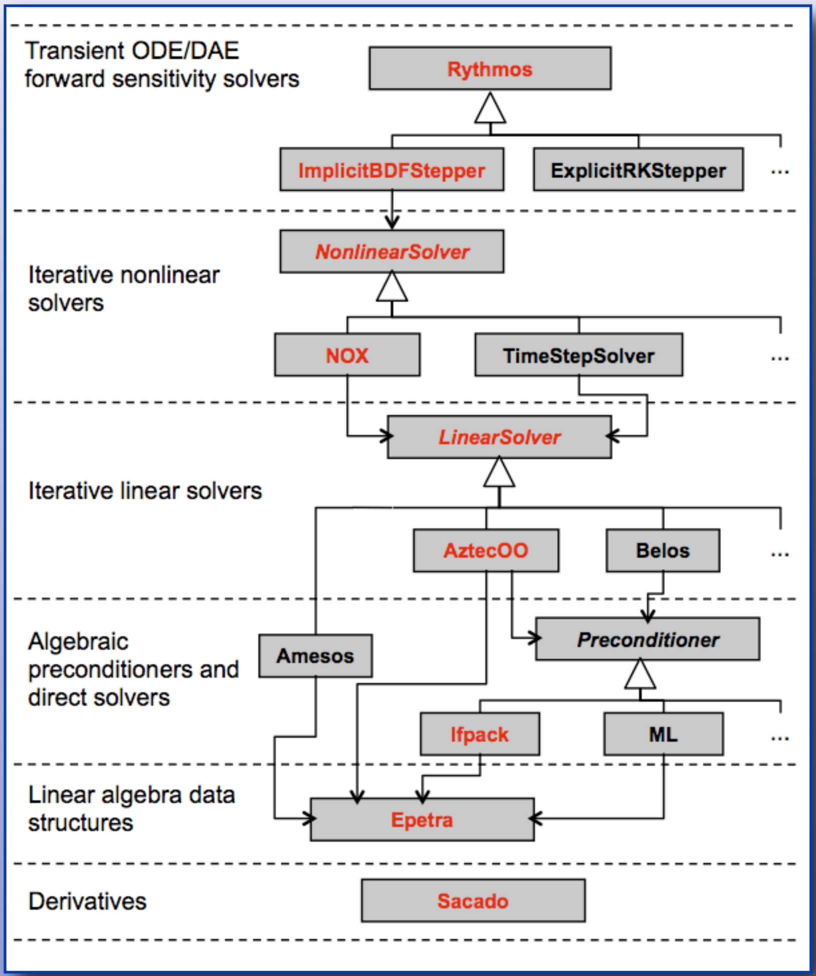
- Solving implicit systems via Newton's method requires vary large linear system solves



$$\text{size} = \frac{(\text{order} + \text{dim})!}{(\text{order})! \cdot (\text{dim})!} \cdot \text{spatial dim}$$

Summary

- Templating key to AD approach
  - Simple, fast Sacado AD tools
  - Apply at "element" level
  - Hooks for future program transformation
- Vertical integration of Trilinos technologies provide remarkable capabilities
  - Efficient, accurate, robust sensitivities
  - Foundation for transient optimization
  - Potential for intrusive stochastic Galerkin methods
  - Requires all levels to be effective


Transient ODE/DAE forward sensitivity solvers
Rythmos
ImplicitBDFStepper
ExplicitRKStepper
...
Iterative nonlinear solvers
NonlinearSolver
NOX
TimeStepSolver
...
Iterative linear solvers
LinearSolver
AztecOO
Belos
...
Algebraic preconditioners and direct solvers
Amesos
Preconditioner
Ifpack
ML
...
Linear algebra data structures
Epetra
Derivatives
Sacado